**Code Coverage and Mutation Testing Report**

**Prepared by:** Yeswanth Nidamanuri

**Project Name:** Car Retail System

**File**: car\_retail\_system.py

**Date:** 12/09/2024

**GitHub:** <https://github.com/yeswanthchowdarynidamanuri/car_rental_management.git>

**Introduction**

The testing strategies and the outcomes identified in the tests of the test\_car\_retail.py module will be outlined in this report since it is a significant subsect of the car retail system. Conducted testing for unit testing, code coverage, and mutation testing to assess the reliability, easy maintainability and robustness of the codes. Below is an overview of the tools and techniques employed:

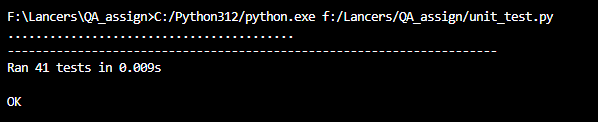
* **Unit Testing:** Built on the unittest framework which is a common generic unit testing framework in Python to cost effectively test diverse individual functions and classes. In order to catch the bugs early and make sure that the modular behaviour in the function is meeting requirements it is important to have unit tests.
* **Code Coverage Analysis:** Applied the tool called coverage.py to know how much of the code is covered by the test suite. Low code density speaks of the fact that a small sample of code is being tested, thereby fortifying confidence in the software. Also, it saves code review effort on areas which have not been tested yet and thus provides overall reduction of review overhead.
* **Mutation Testing:** It was also carried out using mutation testing with Mutatest where mutations are made to the existing code. The efficiency of the test suite is calculated with reference to its ability to expose all of these mutations. Higher mutation score implies that the test suite is able to detect potential faults in order to provide correctness to the code.

**Testing Steps and Process**

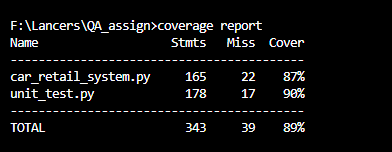
1. **Unit Test Execution:**
   * All unit tests was performed to ensure the proper working of specific subroutines included in the test\_car\_retail.py file.
   * For each test, I concentrated on definite methods or classes; therefore, the behaviors matched the expected outcomes.
2. **Code Coverage Analysis:**
   * Code coverage was then determined using a tool known as coverage.py.
   * It helped in understanding how the test suite is utilizing the code and areas of how the code is not tested at all.
   * The rationale behind increasing coverage is the belief that once more tests achieve coverage then the code is more likely to be bug-free.
3. **Mutation Testing:**
   * Performed intentional (baseline) code modifications by following the Mutatest approach.
   * It was also important to run the test suite in order to observe its capacity to identify and fail on such mutations.
   * Calculated the mutation score so as to understand if the test suite is capable of detecting faults or not.
   * Continually reviewed the test suite with a view of augmenting the test set ability to detect the mutated statements and at the same time enhancing the total mutation score.

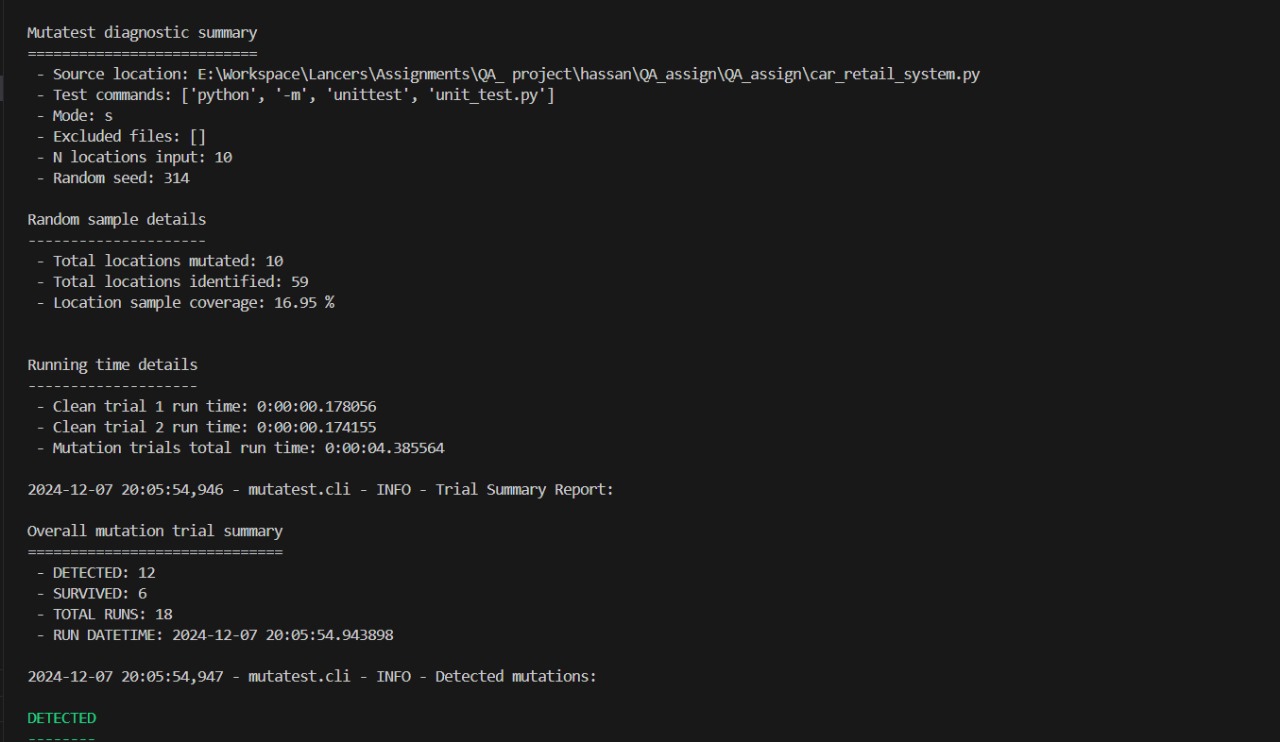
**Tests Result:**

**Unit Test Result:**

****

**Coverage report:**

**Mutation Testing:**

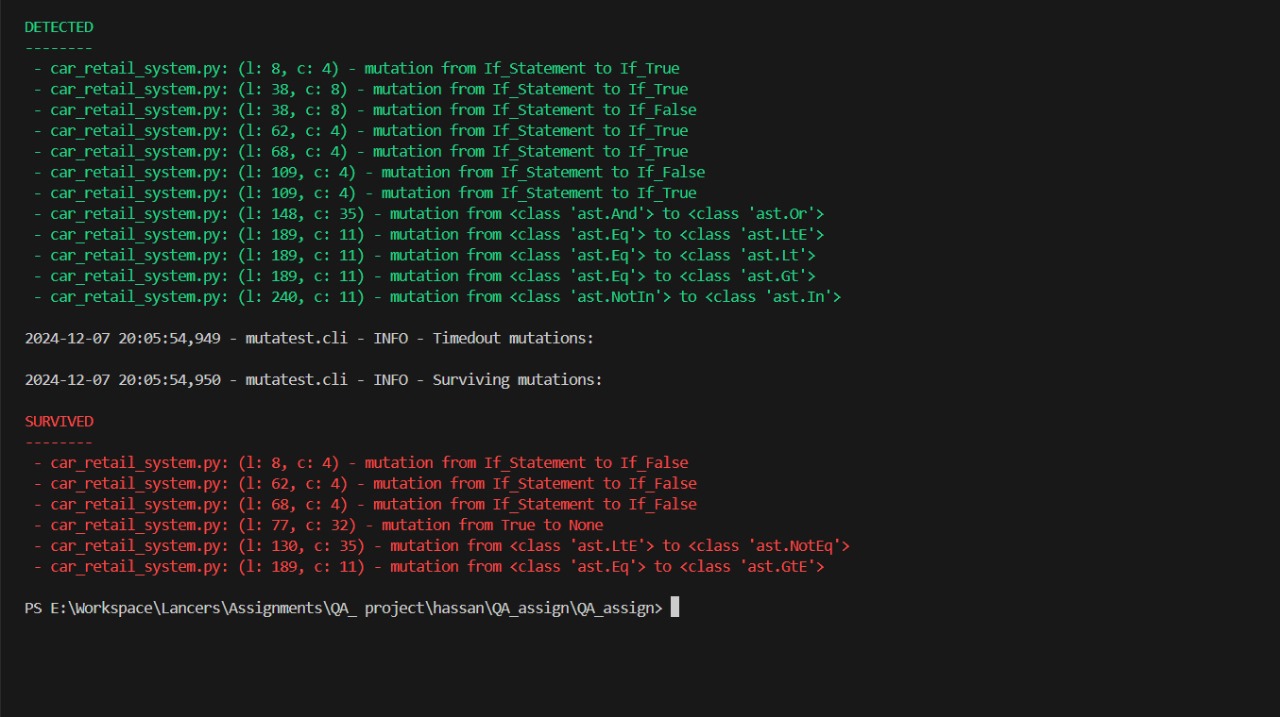


**Result:**

**Killed:** 123  **Timed Out:** 5 **Survived:** 22

* **Killed Mutations:** All these mutations were detected by the test suite hence proving that the test cases developed are capable of detecting faults on the code. This is a good effect on the whole since it proves that the tests are quite stable and can detect possible bugs assuming that they can be produced by minor changes.
* **Timed-Out Mutations:** These mutations were not processed within the global time constraint of the mutation testing tool. This may happen whereby the algorithm has to perform certain activities that require lot of inputs, or whenever the application has some algorithmic tests that involve very complex adjustments. Nowhere is this more obvious than on timeout issues; even though, they may not principally relate to the specific quality of a test, they could entail issues of efficiency in resource utilization or test design approaches.
* **Survived Mutations:** It was also found that some of these mutations were not caught by the test suite, indicating that there may be important areas of the programme that have not been fully tested or where the testing itself may not have been rigorous enough. Remaining mutations correspond to potential weaknesses in the actual code as the test suite did not identify the changes in the code. To this end, the test cases used should be elaborated, and much more flexible to capture any small changes and further promote code robustness and quality.

**Surviving and Detected Mutations:**



**Conclusion**

This report outlines the use of automated testing strategies, code coverage analysis and mutation testing, on a car retail system coded in Python having about 250 lines of code. The results point out seriousness of testing in retaining high quality software.

**Key Findings**

1. **Code Coverage**
   * The evaluation showed that present test coverage is high at 89% of the code.
   * This indicates strong baseline above which the remaining 11% left uncovered might be harbouring some risks. Extending the testing area into these regions would also afford an even greater improvement on code overall integrity.
2. **Mutation Testing**
   * With mutation testing it was shown that many of the introduced mutations are detected effectively by the test suite.
   * Nevertheless, many mutations were not eliminated meaning there are holes in the test suite with regard to fault identification. These missed mutations point out the areas that either the coverage by the tests or the quality of tests that is being used needs an enhancement in order to improve the chances of adequately containing and eradicating all bugs.

**Recommendations for Improvement**

1. **Increase Test Coverage:** There is a tremendous focus on finding more test cases that will help to achieve the best results while covering the remnants of the program, which was determined as the critical area during mutation testing.
2. **Enhance Test Quality:** Improve the used test cases to make it able to locate different kind of faults from the initial ones that where detected. Utilize better and stronger propositions which will actually help in the detection of a fault better.
3. **Integrate Regular Mutation Testing:** First of all, in order to implement the plan we need to take into account high-risk areas and specific regions where the main points of our concentration will be located. To perform the testing efficiently, it is important to identify and select the parts where the testing is most crucial as those elements are critical to the system, or the areas where the issues are most likely to appear.
4. **Make Mutation Testing a Part of Their Daily Affair:** Integrate mutation testing into the development process as the standard tool. It then serves the purpose of enabling one to identify and fix problems in the test suite before they snowball out of control.
5. **Employ Advanced Testing Techniques:**

Look into advanced reports such as:

* **Fuzz Test:** To expose weaknesses through feeding unwarranted or random data.
* **Model-Based Testing:** To create test cases from figures of the system, thus enhancing test coverage.

**Conclusion**

The analysis shows that the comprehensive and systematic testing practices likely lead to high-quality software. However, though the current testing suite is very strong, adding covered aspects and refining mutation testing results further increases reliability and maintainability. It is, therefore, believed that implementing recommendations in this report will leave the car retail system with a robust, fault-tolerant, and future-proof codebase.